This is the final peer-reviewed accepted manuscript of:

Montagna F., Tagliavini G., Rossi D., Garofalo A., Benini L. (2021) Streamlining the OpenMP Programming Model on Ultra-Low-Power Multi-core MCUs. In: Hochberger C., Bauer L., Pionteck T. (eds) Architecture of Computing Systems. ARCS 2021. Lecture Notes in Computer Science, vol 12800. Springer, Cham.

The final published version is available online at: **https://doi.org/10.1007/978-3-030-81682-7_11**

Rights / License:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/)

When citing, please refer to the published version.

Streamlining the OpenMP Programming Model on Ultra-Low-Power Multi-Core MCUs

Fabio Montagna[1], Giuseppe Tagliavini[1], Davide Rossi[1], Angelo Garofalo[1], and Luca Benini[1,2]

[1] University of Bologna, Bologna, Italy
[2] ETH Zürich, Zürich, Switzerland

Abstract. High-level programming models aim at exploiting hardware parallelism and reducing software development costs. However, their adoption on ultra-low-power multi-core microcontroller (MCU) platforms requires minimizing the overheads of work-sharing constructs on fine-grained parallel regions. This work tackles this challenge by proposing OMP-SPMD, a streamlined approach for parallel computing enabling the OpenMP syntax for the Single-Program Multiple-Data (SPMD) paradigm. To assess the performance improvement, we compare our solution with two alternatives: a baseline implementation of the OpenMP runtime based on the fork-join paradigm (OMP-base) and a version leveraging hardware-specific optimizations (OPM-opt). We benchmarked these libraries on a Parallel Ultra-Low Power (PULP) MCU, highlighting that hardware-specific optimizations improve OMP-base performance up to 69%. At the same time, OMP-SPMD leads to an extra improvement up to 178%.

Keywords: ultra-low-power multi-core MCU· parallel programming · OpenMP · SPMD

1 Introduction

In recent years, ultra-low-power (ULP) multi-core microcontroller units (MCUs) have been introduced in low-cost, low-power IoT end-nodes and embedded systems markets [21] [22]. These platforms can provide more than one order of magnitude increase in energy efficiency with respect to high-performance single-core MCUs and carry out the computational power to support the execution of complex workloads. As a representative of this class of MCUs, the PULP platform [21] is an open-source, scalable, and energy-efficient multi-core architecture tailored for sub-mW deeply embedded applications and IoT end-nodes.

Effective programming of these architectures requires the adoption of high-level parallel programming models. However, to achieve high efficiency, we need to tune runtimes to their specific characteristics and tightly limited resources. First, exploiting the ULP features of the hardware architecture can lead to an efficient implementation of the programming model. For instance, the PULP platform includes specialized hardware for accelerating key parallel

patterns (e.g., barriers and locks). Second, parallel programming models imply unavoidable overheads to distribute the workload and orchestrate communication/synchronization among the workers. The overhead minimization in the case of fine-grained parallelism is a key challenge on these platforms. For instance, typical applications have small working sets implying relatively small parallel regions (just a few tens of cycles), making it difficult to amortize overheads.

Fork-join parallelization and Single-Program Multiple-Data (SPMD) are two common paradigms in parallel programming. In the fork/join paradigm, the program execution starts with a single thread, exploiting parallelism recruiting additional threads when a parallel region is encountered (*fork*). When the parallel region ends, only the initial thread continues the program execution sequentially after synchronization (*join*). A well-known programming model based on the fork/join paradigm is OpenMP [9] [10], which allows exploiting parallelism through directives resolved at compile-time into low-level calls for a specific runtime library. With the SPMD paradigm, all the cores start the program execution simultaneously. CUDA and OpenCL programming models adopt this approach on Single-Instruction Multiple-Thread (SIMT) hardware platforms such as GPUs. However, the CUDA support is specific to NVidia platforms, while OpenCL is more portable but requires a total code refactoring. In the domain of embedded systems, it is a common practice to have low-level libraries providing SPMD-compliant primitives providing core identification and synchronization [14] [6] [24].

In this work, we propose a novel approach based on the SPMD paradigm, leveraging the intuitive front-end of the OpenMP programming model to hide the increase in code complexity. Moreover, we present a comprehensive comparison to assess the benefits of the proposed approach. We compared two variants of the OpenMP runtimes: the first one (*OMP-base*) is a baseline implementation for an embedded MCU-class target, with the aim to reduce the code footprint and the execution time; the second one (*OMP-opt*) is fully optimized to take advantage of the PULP hardware support for core idling and synchronization. We considered a set of Digital Signal Processing (DSP) kernels and a full application that are highly representative of the embedded DSP domain and cover a wide range of typical parallelization schemes.

The main contributions of this paper are:

- the proposal of dedicated compile-time transformations to hide the increase in code complexity deriving by adopting the low-level SPMD runtime, using the OpenMP directives as a front-end (*OMP-SPMD*).
- a comprehensive comparison between the *OMP-base* runtime, a optimization of the OpenMP runtime (*OMP-opt*) for an embedded target yet preserving the fork/join behavior, and the proposed approach (*OMP-SPMD* - preserving the OpenMP syntax);
- a comparison between two different programming paradigms (fork/join and SPMD) in the domain of ULP parallel MCUs, revealing that the *OMP-SPMD* approach leads to performance improvement in terms of execution

time and energy consumption of up to 178% compared to the native fork/join approach adopted for OpenMP.

We performed the experiments on a prototype implementation of the open-source PULP platform targeting a cycle-accurate FPGA emulator. Experimental evidence highlights that the *OMP-opt* runtime improves performance up to 69% w.r.t. to the baseline and up to 178% w.r.t. the proposed *OMP-SPMD* runtime.

2 Related work

Parallel programming models provide abstractions to execute applications over multi- and many-core computing platforms [5]. They differ for many aspects (e.g., data organization, workload distribution, scheduling, communication, and synchronization), which imply a trade-off between full transparency for the programmer and rewriting the code from scratch. OpenMP is a widespread programming model for shared-memory platforms, and it has already been demonstrated in the context of embedded systems. In this domain, a common solution is to re-implement its semantics on top of resource-constrained middleware or even bare-metal [12] [3]. The main programming models for general-purpose computing on graphics processing units (GPGPU) computing (i.e., CUDA [14] and OpenCL [6]) are based on the SPMD paradigm. While CUDA is specific for NVidia platforms, OpenCL has been adopted in the domain of heterogeneous embedded systems [25]. In the context of homogeneous multi-core systems, the adoption of SPMD over Multiple-Program Multiple-Data (MPMD) can be beneficial to increase the speed-ups on homogeneous multi-core systems due to factors such as the improved locality for code and data, the reduction of the memory footprint and synchronization overheads [23] [4] . In this work, we compare the two paradigms to understand which one is more suitable for the multi-core MCU target, considering the main factors impacting this investigation.

The lowering of OpenMP directives consists of a set of code transformations that collect the affected code into new functions (outlining) and insert calls to runtime functions. In the typical case, this process is performed by compilers at an early stage of the front-end (e.g., GCC and clang/LLVM). This approach allows a runtime designer to map OpenMP directives into runtime calls, implementing other paradigms w.r.t. fork/join. Pereira *et al.* [16] describe a framework that automatically converts program sections annotated with OpenMP 4.x directives into OpenCL kernels. This design goes in the direction of our work, but we perform a step further, considering the severe constraints of the ULP MCUs, requiring specific optimizations.

The OpenMP specification includes a `simd` directive that can be applied to a loop with the intent to map multiple iterations on a set of instructions exposing SIMD semantics. For instance, this construct can be used to exploit the packed-SIMD instructions available on our target platform to perform operations on vectors with 8-bits or 16-bits elements. However, our approach based on the SPMD paradigm is more general as it is not limited to a program part (i.e., loops) but enables a global optimization of the whole program.

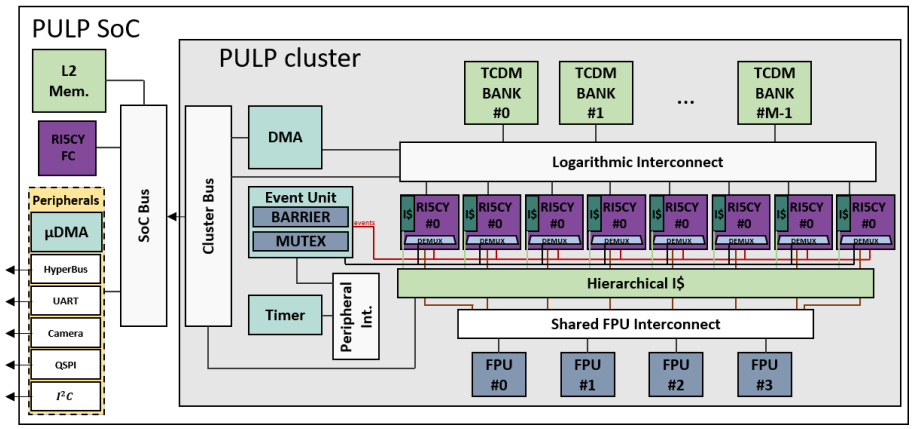

Fig. 1. Top-level view of the PULP architecture.

3 Background

3.1 PULP platform

In this paper, we target a ULP multi-core MCU based on the open-source PULP platform. Using the open-source RTL, we instantiated a cycle-accurate emulation image of a multi-core cluster on a Xilinx VCU118 FPGA [19]. PULP is a multi-core programmable processor that features a RISC-V based core for control functions and a cluster (Fig. 1) of 8 RISC-V based cores for energy-efficient DSP. These cores, namely RI5CY [7], implement a 4 stage in-order single-issue pipeline, supporting the RV32IMC instruction set, plus extensions for optimized DSP and machine learning [18]. The platform features two memory levels, a 512 kB L2 memory (15 cycle latency for load/store operations) outside the cluster and a single cycle latency, multi-bank, 64 kB L1 memory inside the cluster, which enables shared-memory parallel programming. The cluster also includes four floating-point units (FPUs) shared among the cluster cores. The *event unit* (EU) is a hardware block introduced to support fine-grained parallelism with minimum overhead [8]. This unit accelerates the execution of data-parallel patterns (e.g., thread dispatching, barrier semantic, and critical regions) and enables power-saving policies to put the unused cores in idle state.

3.2 Parallel Programming Paradigms

OpenMP is one of the most adopted high-level programming models in different computing domains, from High-Performance Computing (HPC) to embedded systems. It makes use of directives (defined for C/C++ and Fortran languages), which are resolved at compile-time into low-level calls for a dedicated runtime library, such as GNU libgomp [9] or LLVM OpenMP library [10].

OpenMP relies on a fork/join parallel execution model. The execution of the program starts with a single thread (called *master*). When a parallel construct is encountered, $n-1$ additional threads (*workers*) are recruited into a parallel team. *Work-sharing constructs* are employed to specify how the parallel workload is distributed among the threads. When the parallel region ends, all the threads reach a barrier for synchronization. Then, the master thread continues its execution sequentially.

The second approach considered in this paper adopts the SPMD paradigm, using a set of primitives for control flow handling and inter-core synchronization. In contrast to the fork/join paradigm adopted by OpenMP, where only the master core starts executing the program until the execution flow encounters a parallel region, all the cores start executing the same code. The cores follow the same execution flow unless the programmer explicitly indicates that a subset of cores must execute a specific region; parallel workloads are split among cores (based on the core number) and run concurrently on different data. Moreover, the synchronization points and the allocation of data variables in shared or private areas must be explicit. As the result of a preliminary analysis, the approach based on the SPMD runtime implies more programmer effort than OpenMP since it requires modifying the source code. Nevertheless, its adoption guarantees a higher control on the parallelization process and, in general, less overhead compared to a traditional OpenMP runtime. As introduced in Section 1, the adoption of compile-time code transformations can make this effort totally transparent to programmers. To enable automatic code transformations at compile-time, we need to introduce a set of *SPMD helper functions* that are described in the next section.

4 Deriving the SPMD-OMP model

4.1 Low-level Software Support for Parallel Computing

This work focuses on the work-sharing constructs that are more frequently used to parallelize code in the embedded DSP domain, illustrating the OpenMP directives, the SPMD helper functions, and their mapping. The presented runtimes are based on a common lightweight Hardware Abstraction Layer (HAL), which provides minimal access to platform features in the absence of a full-fledged operating system. This design choice enables a consistent reduction of the overhead and guarantees higher energy efficiency. Getting as much parallelism as possible out from an algorithm is not straightforward and usually requires a significant effort from programmers.

4.2 Work-sharing Constructs

Table 1 reports the OpenMP directives considered in this work and the equivalent SPMD helper functions. In OpenMP, the *#pragma omp for* directive is placed before a loop, informing the compiler that each loop iteration is independent of the others and, thus, executable concurrently. The workload is divided

Table 1. List of the OpenMP directives and SPMD helper functions with a brief usage description.

OpenMP Directives	SPMD Helper Functions	Description
#pragma omp for	SPMD_PARLOOP(start, end, from, to, step)	worksharing construct to distribute loop iterations among threads
#pragma omp for \ schedule(static, chunk)	SPMD_PARLOOP_SCHED(start, end, from, to, step, chunk) SPMD_PARLOOP_STEP(step, chunk)	worksharing construct to distribute loop iterations among threads
#pragma omp for \ reduction(op:var)	SPDM_PARLOOP_REDUCTION_*OP*(var, temp_vars) reduction_func(op_f);	performs a reduction on variable with the operator (op)
#pragma omp master #pragma omp single	IS_CORE_0, ...	identifies a portion of the code executed only by a core
#pragma omp barrier	spmd_barrier();	All the threads wait the other for synchronization
#pragma omp critical	spmd_critical_enter(); spmd_critical_exit();	Critical section

into equal chunks using the static scheduling (default option), where the chunk size is equal to the number of iterations over the number of cores involved in the parallel region. There is an implicit barrier at the end of the loop (unless a *nowait* clause is specified). In the SPMD programming model, the helper macro *SPMD_PARLOOP* computes the loop bounds (based on chunk size and core id) used to distribute the iterations on the available cores. In this case, barrier semantics must be explicit (e.g., calling *spmd_barrier*). The *#pragma omp for* directive also provides a *schedule(static, chunk)* clause to specify a custom chunk size. In SPMD, we can obtain the same behavior using the helper macros *SPMD_PARLOOP_SCHED* and *SPMD_PARLOOP_STEP* to compute the loop bounds (start and end) and the iteration step.

OpenMP supports loop reductions using the *reduction(op:var)* clause. A shared variable performs an accumulation based on a standard operator (i.e., +, -, *). In this case, OpenMP runtimes adopt a mechanism to avoid race conditions due to multiple accesses on the shared variable from multiple cores. In SPMD, a shared array must be explicitly declared before calling the helper function. In this way, each core can store the intermediate accumulation values into a dedicated array element (usually the one corresponding to the core id). At the end of the computation, the master core accumulates all the partial results in the target variable. A set of macros *SPMD_PARLOOP_REDUCTION_OP* perform the final reduction step based on the used operator.

Most of the algorithms are not fully parallel and, hence, include sequential code regions. The *#pragma omp master/single* directives allow executing the sequential code with a single core (the master or a generic one, respectively). This directive does not feature an implicit barrier; thus, a synchronization point must be added (when needed). In the SPMD approach, the code is enclosed in an *if* block that can be accessed only by one core (e.g., the core with the id equal to 0), while the others continue the execution or can be blocked on a barrier. The id of the executing core can be checked with a set of Boolean preprocessor macros (*IS_CORE_0*, ...).

The *#pragma omp critical* OpenMP directive specifies a portion of code that must be executed from one core at a time. This directive adds a total order constraint and, consequently, reduces the program speed-up. In SPMD, we can specify a critical section enclosing a region of code between the *spmd_critical_enter* and *spmd_critical_exit* helper functions. Finally, the *#pragma omp barrier* in OpenMP and *spmd_barrier()* in SPMD synchronize all the cores before proceeding with the rest of the execution. When a core reaches a barrier, it is blocked until all the other cores reach the barrier.

4.3 Event Unit Extensions for Overhead Reduction

The *OMP-opt* runtime makes use of the EU to reduce the overhead of the *OMP-base* directives. The EU design is based on 32 level-sensitive *event lines* (per core) correlated to *event sources*. Two EU extensions, namely *barrier* and *mutex*, contain the logic to handle core-to-core signaling. The *barrier* extension includes a register describing the status of each core. When the core reaches the *barrier*, the matching bit in the status register is set. The EU generates an event when all the cores reach the barrier, interpreted as a continuation condition for the idle cores. The *mutex* extension enables mutual exclusivity supporting synchronization primitives, being a resource that can only be owned by one core at a time. Tentative accesses (try-lock semantic) are signaled in a dedicated status register. The *mutex* extension keeps track of all pending requests.

The *OMP-opt* runtime leverages the EU to reduce the overhead associated with parallel regions and barrier constructs. Opening a parallel region, the *mutex* extension enables fast and mutually exclusive access to shared data structures, and the *barrier* extension simplifies the creation of a team of threads. When encountering a barrier construct, the *barrier* extension provides seamless support for the synchronization semantic. The benefits of hardware support are evident in OpenMP-based applications employing a higher amount of work-sharing and/or synchronization directives. Contrarily, the performance gain is still negligible for embarrassingly parallel kernels.

The introduction of a new runtime based on the SPMD paradigm is justified by the fact that reducing the overhead of the *OMP-base* version through hardware support is usually not enough to approach the ideal performance of a benchmark. The intuition for this effect is related to the granularity of parallel code regions. In the case of fine-grained parallelism, the overhead required to create multiple parallel teams can be significant. Also, the overhead for the loop bound computation in different parallel regions can be reduced by applying common subexpression elimination (CSE), which is a standard optimization pass in compiler toolchains. However, the code outlining for different regions in a standard fork/join runtime can make its application harder.

4.4 Mapping OpenMP directives on the SPMD paradigm

As a motivating example to explain our approach, Fig. 2 shows the use of the OpenMP directives (left) and the SPMD helper functions (right) to parallelize

```
void kmeans()
{
  // ...
  do {
    delta = 0.0f;
    #pragma omp parallel \
          num_threads(NUM_CORES) \
          shared(delta) private(index)
    {
      /* main computation */
      #pragma omp for reduction(+: delta)
      for (i=0; i<N_OBJECTS; i++) {
        // ...
      }

      /* array reduction */
      #pragma omp for nowait
      for (i=0; i<N_CLUSTERS; i++) {
        for (j=0; j<NUM_CORES; j++) {
          // ...
        }
      }
      // ...
    }
  } while (delta > THRESH &&
           loop < MAX_ITERS);
  // …
}
```

```
void kmeans()
{
  if(core_id == 0)
  {
    // …
  }

  /* Compute loop bounds */
  SPMD_PARLOOP(start, end, 0, N_OBJECTS);
  SPMD_PARLOOP(start2, end2, 0, N_CLUSTERS);

  do {
    local_delta[core_id] = 0;
    /* main computation */
    for (i=start; i<end; i++) {
      // ...
    }
    spmd_barrier();
    /* Reduction on delta */
    SPMD_PARLOOP_REDUCTION_SUM(delta,
                               local_delta);
    /* array reduction */
    for (i=start2; i<end2; i++) {
      for (j=0; j<NUM_CORES; j++) {
        // ...
      }
    }
    // ...
  } while (delta > THRESH &&
           loop < MAX_ITERS);

  if(core_id == 0)
  {
    // ...
  }
}
```

Fig. 2. Code snippet that shows a simple use of OpenMP directives (left) and SPMD helper functions (right) applied to the K-MEANS kernel.

the K-MEANS kernel, included in the benchmark suite described in Section 5. The OpenMP version is characterized by an overhead associated with opening parallel regions. State-of-the-art solutions provide a dedicated OpenMP runtime optimized for the embedded target, intending to reduce overheads as much as possible, even not supporting some features that are considered unnecessary. However, the overhead of parallel regions cannot be reduced under a few hundred cycles. Considering the example depicted in Fig. 2, a parallel directive inside a loop can incur a significant overhead.

In the SPDM version, the code outside parallel regions requires an additional check to force sequential execution, but the overhead of this operation is negligible (maximum 3 cycles). The additional code (highlighted in bold) is functionally equivalent to the code produced by the compiler for OpenMP, so it is not a source of overhead. Moreover, additional code optimizations are possible for the computation of static loop bounds because they are in the same code block. The SPMD interface requires to provide additional parameters (*start* and *end* indices), while OpenMP totally hides these details. The transformation from

OpenMP-annotated code to its SPMD variant is syntactically well-defined; consequently, it can be performed as a source-to-source translation or a direct modification to the data structures in the compiler front-end. From this perspective, Table 1 provides a map to translate the OpenMP directives into an equivalent code adopting the SPMD helper functions. In addition to these guidelines, an additional requirement derives from the main difference between for/join and SPMD paradigms: the code outside an OpenMP parallel region must be executed by core 0. This behavior can be easily enforced by adding a conditional statement to the code regions that are not annotated.

In this work, we adopted a source-to-source approach. We prototyped our methodology using ROSE [20], an open-source tool developed at Lawrence Livermore National Laboratory to enable source-to-source program analysis and transformation. ROSE produces a high-level representation of the source code based on an abstract syntax tree (AST). It provides an API to analyze and modify the AST representation to derive a modified source code. We modified the standard ROSE flow for lowering OpenMP directives as follows:

- visiting the AST structure, the code outside a parallel region is enclosed by an if statement to ensure sequential execution;
- the computation of the loop bounds for the parallel loops is performed using the SPMD helper macros, which are placed at the beginning of the helper function to promote optimizations such as CSE;
- barriers and critical regions are mapped on the related SPMD functions, replacing the call to libgomp functions.

This transformation flow is automatically applied to the OpenMP program, and any additional modification is required to the programmer. The current prototype supports the OpenMP directives used by the benchmarks, which are reported in Table 2. Future extensions are discussed in Section 8.

5 Benchmarks

To compare the runtimes on the target multi-core architecture, we evaluate the performance of eight benchmarks that are commonly used in DSP for feature extraction, classification, and basic linear algebra functions. Table 2 reports the OpenMP directives used to parallelize the benchmarks, their application domains, and the percentage of parallelizable code.

The Principal Component Analysis (PCA) [1] is used for compression and feature extraction. It performs an orthogonal transformation, mapping possibly correlated variables into a set of linearly uncorrelated components. It requires a mix of directives (Table 2), called multiple times, to exploit parallelism, resulting in a complex parallel scheme. Another common kernel used for feature extraction is the Discrete Wavelet Transform (DWT) [11], which decomposes a signal into a different level of frequency resolutions through a bank of Low Pass (LPF) and High Pass Filters (HPF), capturing both temporal and frequency information, easily parallelizable using *#pragma omp for* directives and explicit

Table 2. For each benchmark, this table reports the main application domain (Domain), the OpenMP directives applied (OpenMP Dir.) and the percentage of the parallelizable code (Par. Code[%]).

Application	Domain	OpenMP Dir.	Par.Code[%]
CONV	Audio, Image, ExG	#pragma omp for	100
DWT	Audio, Image, ExG	#pragma omp for #pragma omp for schedule(static, chunk) #pragma omp barrier	100
FFT	Audio, Image, ExG	#pragma omp for #pragma omp barrier	100
MATMUL	Audio, Image, ExG	#pragma omp for	100
PCA	ExG	#pragma omp for #pragma omp for reduction(var:oper) #pragma omp master #pragma omp barrier	95
SVM	ExG	#pragma omp for #pragma omp master #pragma omp barrier	99
K-MEANS	Audio, Image, ExG	#pragma omp for #pragma omp master #pragma omp barrier	97
M-NORM	Audio, Image, ExG	#pragma omp for #pragma omp for reduction(var:oper)	100

barriers. The Fast Fourier Transform (FFT) [2] transforms a signal from the time domain to the frequency domain. The cores of the cluster work on different data, enforcing consistency with synchronization barriers. There are several variants of this algorithm; in this paper, we consider decimation-in-frequency radix-2. The Support Vector Machine (SVM) [15]is a classifier that is widely used in machine learning embedded applications. Starting from a set of support vectors (SVs) that compose a hyper-plane, it classifies unknown samples into a known class, exploiting parallelism using *#pragma omp for* and barriers. We also include an unsupervised classifier named K-Means, which can inference an unknown outcome from input vectors. The parallel scheme includes *#pragma omp for*, reduction directives, and sequential sections. The Mean-Normalization (M-Norm) is a widespread operation in Machine Learning, and it is used to transform the data such that the new output vector has zero-mean. The parallelism is exploiting using *#pragma omp for* and reductions. The last two kernels are Basic Linear Algebra Subprograms (BLAS) commonly used in DSP: matrix multiplication (MatMul) and convolution (Conv), which is the most computing-intensive kernel in Convolutional Neural Network (CNN) workload. Both of them are fully parallel, requiring *#pragma omp for* directives to split the workload among the cores and synchronization barriers.

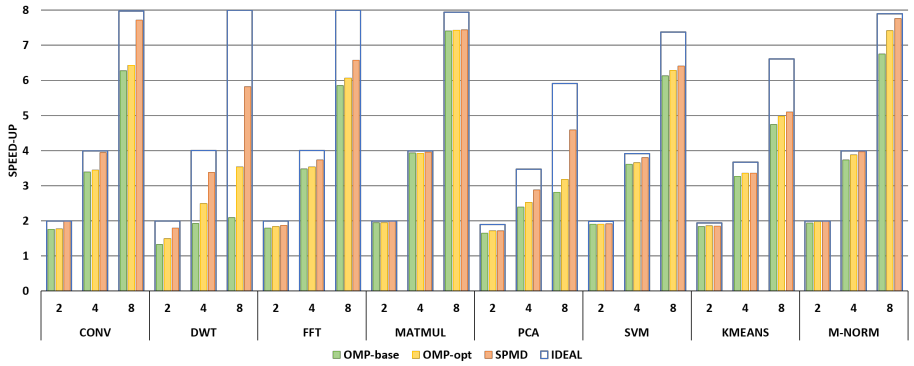

Fig. 3. Speed-ups from the parallel execution of the eight DPS benchmarks using the OMP-base, OMP-opt, and *OMP-SPMD* runtimes.

6 Experimental Results

In this section, we present an experimental assessment executing the benchmark suite on the FPGA emulator described in Section 3.1. We used the hardware performance counters available on the PULP cores, which provide accurate metrics on the core operation (clock cycles, executed instructions, instruction cache misses, memory contentions, pipeline stalls, and FPU contentions). We considered the clock cycles required for the benchmark execution, varying the number of cores involved in the computation (up to 8). To evaluate the single-core performance, we considered a version of the code without the overhead from the work-sharing constructs, which are not required for sequential execution. In parallel programs, there is a structural limit to the speed-up given by Amdahl's law. For each benchmark, we computed an *Amdahl limit* by measuring the percentage of parallelizable code (reported in Table 2) and supposing no parallelization overhead. A comparison between the measured speed-ups and the ideal ones provides a quantitative metric for code parallelization.

Fig. 3 depicts a comparison between real speed-ups and Amdahl limits for all runtimes. As expected, *OMP-opt* is closer than *OMP-base* to the theoretical limit, thanks to the adoption of hardware support. To provide insight, Table 3 reports execution times (in cycles, for 1 and 8 cores), gains of the new runtimes over *OMP-base*, numbers of barriers and parallel regions, and the cycles lost for hardware stalls. The stalls derive from different hardware sources: contentions in accessing the TCDM by the cores, instruction cache misses, FPU contentions, and barriers. The stalls for the parallel case consider the *OMP-SPMD* runtime.

The overhead reduction is more evident in the kernels that require more barriers or parallel regions. The *OMP-SPMD* approach further reduces the overhead for two main reasons. First, it does not require the creation of a parallel team corresponding to a parallel construct. Second, a programmer can reduce checks on parallel loop intervals by combining the boundary computations over multi-

Table 3. [a] *OMP-base*, [b] *OMP-opt*, [c] *OMP-SPMD*, [d] [*OMP-base* over *OMP-opt*]/[*OMP-base* over *OMP-SPMD*] runtime (RT)[%], [e] barriers (BR), [f] parallel regions (PR), [g] [1-core stalls (S) / 8-core stalls] in kCycles.

	1 Core	8 Cores						
Kernel	**kCycles**	**kCycles** [a]	**kCycles** [b]	**kCycles** [c]	**RT[%]** [d]	**BR** [e]	**PR** [f]	**S** [g]
CONV	136.9	21.8	21.3	17.7	2 / 23	1	1	0.69/ 0.70
DWT	24.9	11.9	7.0	4.3	69 / 178	27	9	0.28 / 0.55
FFT	228.0	39.0	37.6	34.7	4 / 12	13	1	3.39 / 3.53
MATMUL	959.3	130.4	129.0	127.5	1 / 2	1	1	8.40 / 8.41
PCA	1'173.8	417.4	321.3	255.0	30 / 64	206	262	33.64 / 35.70
SVM	29.6	4.9	4.4	4.3	11 / 14	1	1	3.13 / 3.14
K-MEANS	357.5	75.4	71.9	69.1	5 / 9	189	9	10.27 / 12.16
M-NORM	57.7	8.5	7.8	7.4	10 / 15	3	1	0.04/ 0.01
Application	**kCycles**	**kCycles** [a]	**kCycles** [b]	**kCycles** [c]	**RT[%]** [d]	**BR** [e]	**PR** [f]	**S** [g]
SEIZURE DETECTION	1'230.9	459.6	390.7	256.6	18 / 79	450	344	31.44 / 35.93

ple loops. For the MatMul benchmark, the choice of a specific runtime does not bring to particular improvements, as the three approaches show similar performance (very close to the Amdahl limit). The benchmark that gains the most benefit from SPMD is DWT, which shows a reduction of 69% and 178%, passing from *OMP-base* to *OMP-opt* (i.e., optimized barriers and parallel regions), and finally to *OMP-SPMD* (i.e., optimized control flow), respectively. In DWT, we have reduced the parallelization overhead by unifying the logic to compute the bounds of two parallel loops; the compiler cannot apply the same transformation to the OpenMP version because the loops are in distinct modules of the control flow graph (due to outlining). Consequently, the bounds are computed twice in OpenMP-based runtimes, doubling the overhead.

Varying the cores from 2 to 8, all the benchmarks (except for MatMul) approach the Amdhal limit with the 2-cores execution. For 4 and 8 cores, the speed-ups start saturating as a direct consequence of the overhead implied by the work-sharing constructs. The PCA kernel demonstrates a high benefit in using *OMP-opt* and *OMP-SPMD*. The main reason is that this kernel includes a consistent number of parallel regions and requires several barriers for synchronization. Reducing the overhead of *OMP-base* has a direct impact on performance (i.e., 30% and 64%, respectively). All the other benchmarks, including CONV, FFT, MATMUL, SVM, and M-NORM, show an improvement of up to 23%, 12%, 2%, 14%, and 15%, respectively, which is a direct consequence of the overhead reduction. The only exception is the K-MEANS kernel, which has a limited improvement even with many barriers and parallel regions (up to 9%). In this case, the number of hardware stalls (reported in Table 3) significantly increase with 8 cores, and this effect structurally limits the maximum speed-up. Moreover, the fine granularity of the parallel workload highly reduces the opportunities for optimizations.

Finally, we also evaluated the energy consumption of the DSP benchmarks. The estimation relies on power measurements on a fabricated prototype of Mr. Wolf [17], a PULP architecture with the same features of our FPGA emula-

Table 4. Energy consumption of 8-cores, based on real measurements on a silicon prototype of PULP (operative point: 110MHz@0.8V).

Kernel	**OMP-base** [μJ]	**OMP-Opt** [μJ]	**SPMD** [μJ]	**Reduction** [%]
Conv	3.54	3.49	3.20	1.44 – 9.60
DWT	1.40	1.09	0.77	22.50 – 45.00
FFT	6.61	6.54	6.26	1.06 – 5.30
MatMul	23.27	23.26	23.26	0.01 – 0.04
PCA	75.31	71.79	46.00	4.67 – 38.92
SVM	4.43	4.40	4.34	0.47 – 2.03
K-MEANS	12.99	12.57	12.47	3.20 – 4.00
M-NORM	1.43	1.37	1.34	4.09 – 6.29
Application	**OMP-base** [μJ]	**OMP-Opt** [μJ]	**SPMD** [μJ]	**Reduction** [%]
SEIZURE DETECTION	81.69	77.92	51.72	36.69 – 4.62

tor, running a typical high-utilization workload (matrix multiplication). Table 4 shows the results of the energy consumption for the 8-cores execution at the operating frequency of 110 MHz at 0.8 V, the optimal operating point to maximize energy efficiency. The results show a reduction of the energy consumption by up to 45% passing from *OMP-base* to *SPMD*, while the reduction from *OMP-base* to *Omp-Opt* is capped to 22.50%. The difference between the two gains provides a direct measure of the benefits of the SPMD approach w.r.t. an optimized runtime taking advantage of dedicated hardware units. There is no linear correlation between energy consumption and the variation in parallel speed-up due to the power-savings policy implied by optimizations (e.g., clock-gating).

7 Seizure Detection Application

To better evaluate the impact of our approach in a real scenario, we considered a full application taken from the biomedical field, the seizure detection processing chain [13], which aims at detecting the outcome of a seizure in subjects affected by epilepsy. This application contains three of the benchmarks included in this exploration, more precisely, PCA, DWT, and SVM. As shown in the previous section, these benchmarks demonstrate different parallel schemes and behaviors depending on the chosen runtime. Table 3 presents results in performance and speed-ups. In particular, passing from *OMP-base* to *OMP-opt*, we can see an improvement of 18%, further increased to 79% passing to *OMP-SPMD*, with a speed-up improvement from 2.7× to 4.8×. Table 4 also reports the total energy consumption of this application.

8 Conclusion

In this work, we propose a highly streamlined, low-overhead approach based on the SPMD paradigm as an alternative to the standard OpenMP approach

(based on fork/join) to target the emerging ULP multi-core MCUs. We compared two alternative OpenMP runtimes, a baseline implementation suitable for a generic embedded target and a fully optimized version taking advantage of dedicated hardware support for core idling and synchronization, focusing on the most used work-sharing constructs. We evaluated the performance using a prototype implementation of the PULP platform on an FPGA emulator, using a set of benchmarks from the DSP domain and a full application. We demonstrated that the optimized OpenMP runtime improves performance by up to 69% compared to the baseline; the SPMD approach leads to a further improvement (up to 178%), approaching the maximum achievable speed-up envisioned by Amdahl's law, with an average distance of 18%. The benefits of the *OMP-SPMD* over the *OMP-base* programming model also emerge from the energy consumption, with a gain by up to 82%. To get the best of both worlds, we introduce compile-time source-to-source transformations to hide the increase in code complexity deriving by adopting the low-level *OMP-SPMD* runtime, using the OpenMP directives.

As future work, we will finalize the automatic lowering of the OpenMP directives into SPMD primitives, which is now at a prototype level and lacks support for a subset of OpenMP directives and clauses. We will also evaluate support for advanced directives included in 4.0 and 5.0 OpenMP specifications, even if the dynamic behavior of advanced constructs is generally not suitable for many embedded applications running on MCU-class devices.

Acknowledgments

This work has been partially supported by the European Union's Horizon 2020 research and innovation programme under grant agreement numbers 732631 (OPRECOMP), 863337 (WiPLASH), and 857191 (IOTWINS).

References

1. Abdi, H., Williams, L.J.: Principal component analysis. Wiley interdisciplinary reviews: computational statistics **2**(4), 433–459 (2010)
2. Brigham, E.O.: The fast Fourier transform and its applications. Prentice-Hall, Inc. (1988)
3. Chapman, B., Huang, L., Biscondi, E., Stotzer, E., Shrivastava, A., Gatherer, A.: Implementing OpenMP on a high performance embedded multicore MPSoC. In: 2009 IEEE International Symposium on Parallel & Distributed Processing. pp. 1–8. IEEE (2009)
4. Chen, K.C., Chen, C.H.: Enabling SIMT execution model on homogeneous multi-core system. ACM Transactions on Architecture and Code Optimization (TACO) **15**(1), 1–26 (2018)
5. Diaz, J., Munoz-Caro, C., Nino, A.: A survey of parallel programming models and tools in the multi and many-core era. IEEE Transactions on Parallel and Distributed systems **23**(8), 1369–1386 (2012)
6. Gaster, B., Howes, L., Kaeli, D.R., Mistry, P., Schaa, D.: Heterogeneous computing with OpenCL. Newnes (2012)

7. Gautschi, M., Schiavone, P.D., Traber, A., Loi, I., Pullini, A., Rossi, D., Flamand, E., Gürkaynak, F.K., Benini, L.: Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **25**(10), 2700–2713 (2017)
8. Glaser, F., Tagliavini, G., Rossi, D., Haugou, G., Huang, Q., Benini, L.: Energy-Efficient Hardware-Accelerated Synchronization for Shared-L1-Memory Multiprocessor Clusters. IEEE Transactions on Parallel and Distributed Systems **32**(3), 633–648 (2021)
9. GNU Foundation: libgomp runtime, https://gcc.gnu.org/onlinedocs/libgomp/
10. LLVM Project: LLVM OpenMP runtime, https://openmp.llvm.org/Reference.pdf
11. Mallat, S.G.: A theory for multiresolution signal decomposition: the wavelet representation. IEEE Transactions on Pattern Analysis and Machine Intelligence **11**(7), 674–693 (1989)
12. Mitra, G., Stotzer, E., Jayaraj, A., Rendell, A.P.: Implementation and optimization of the OpenMP accelerator model for the TI Keystone II architecture. In: Using and Improving OpenMP for Devices, Tasks, and More. pp. 202–214. Springer (2014)
13. Montagna, F., Benatti, S., Rossi, D.: Flexible, scalable and energy efficient biosignals processing on the pulp platform: A case study on seizure detection. Journal of Low Power Electronics and Applications **7**(2), 16 (2017)
14. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA: Is cuda the parallel programming model that application developers have been waiting for? Queue **6**(2), 40–53 (2008)
15. Noble, W.S.: What is a support vector machine? Nature biotechnology **24**(12), 1565–1567 (2006)
16. Pereira, M.M., Sousa, R.C., Araujo, G.: Compiling and optimizing OpenMP 4. X programs to OpenCL and SPIR. In: International Workshop on OpenMP. pp. 48–61. Springer (2017)
17. Pullini, A., Rossi, D., Loi, I., Tagliavini, G., Benini, L.: Mr. Wolf: An energy-precision scalable parallel ultra low power SoC for IoT edge processing. IEEE Journal of Solid-State Circuits **54**(7), 1970–1981 (2019)
18. PULP Project: RI5CY Manual, https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf
19. PULP Project: Setup of Xilinx FPGA boards, https://github.com/pulp-platform/pulp/tree/master/fpga/pulpissimo-zcu104
20. Quinlan, D.: ROSE: Compiler support for object-oriented frameworks. Parallel processing letters **10**(02n03), 215–226 (2000)
21. Rossi, D., Conti, F., Marongiu, A., Pullini, A., Loi, I., Gautschi, M., Tagliavini, G., Capotondi, A., Flatresse, P., Benini, L.: PULP: A parallel ultra low power platform for next generation IoT applications. In: 2015 IEEE Hot Chips 27 Symposium (HCS). pp. 1–39. IEEE (2015)
22. Sony Corporation: Sony Spresense multicore microcontroller, https://developer.sony.com/develop/spresense/
23. Stratton, J.A., Grover, V., Marathe, J., Aarts, B., Murphy, M., Hu, Z., Hwu, W.m.W.: Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs. In: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization. pp. 111–119 (2010)
24. Tagliavini, G., Cesarini, D., Marongiu, A.: Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight openmp tasking. IEEE Transactions on Parallel and Distributed Systems **29**(9), 2150–2163 (2018)
25. Taylor, B., Marco, V.S., Wang, Z.: Adaptive optimization for OpenCL programs on embedded heterogeneous systems. ACM SIGPLAN Notices **52**(5), 11–20 (2017)